Experimental Evaluation of Bounded-Depth LSM Merge Policies

Qizhong Mao[1], Steven Jacobs[2], Waleed Amjad[3], Vagelis Hristidis[4] Vassilis J. Tsotras[5], Neal E. Young[6]
Computer Science and Engineering, University of California, Riverside, USA
Email: { [1]qmao002, [2]sjaco002, [3]wamja001, [4]evangelo, [5]vtsotras, [6]neal.young }@ucr.edu

***Abstract*—Modern NoSQL databases use log-structured merge (LSM) storage architectures to support high write throughput. LSM architectures aggregate writes in a mutable *MemTable* (stored in memory), which is regularly flushed to disk, creating a new immutable file called an *SSTable*. Periodically, some of the SSTables are chosen to be *merged* — replaced with a single SSTable containing their union. A *merge policy* (a.k.a. compaction policy) specifies when to do merges and which SSTables to combine. A *bounded depth* merge policy is one that guarantees that the number of SSTables never exceeds a given parameter k, typically in the range 3–10. Bounded-depth policies are useful in applications where low read latency is crucial, but they and their underlying combinatorics are not yet well understood. This paper compares several bounded-depth policies, including representative policies from industrial NoSQL databases and two new ones based on recent theoretical modeling. The results validate the proposed theoretical model and show that, compared to the existing policies, the newly proposed policies can have substantially lower write amplification.**

***Index Terms*—NoSQL, LSM, merge policy, compaction**

I. Introduction

Modern NoSQL systems [1], [2] use log-structured-merge (LSM) architectures [3] to achieve high write throughput. To insert a new record, a WRITE operation simply inserts the record into the memory-resident *MemTable* [2] (also called the in-memory component). UPDATE operations are implemented lazily, requiring only a single WRITE to the MemTable. DELETE operations are implemented similarly, by writing an *anti-matter* record for the key to the MemTable. Thus, each WRITE, UPDATE, or DELETE operation avoids any immediate disk access. When the MemTable reaches its allocated capacity (or for other reasons), it is flushed to disk, creating an immutable disk file called a component, or, usually, an *SSTable* (Sorted Strings Table [2]). This process continues, creating many SSTables over time.

Each READ operation searches the MemTable and SSTables to find the most recent value written for the given key. With a compact index stored in memory for each SSTable, checking whether a given SSTable contains a given key typically takes just one disk access [4, §2.5]. (For small SSTables, this access can sometimes be avoided by storing a Bloom filter for the SSTable in memory [5].) Hence, the time per READ grows with the number of SSTables. To control READ costs, the system periodically *merges* SSTables to reduce their number and to prune updated and anti-matter records. Each merge replaces some subset of the SSTables by a single new SSTable that holds their union. The merge batch-writes these items to the new SSTable on disk. The *write amplification* is the number of bytes written by all merges, divided by the number of bytes inserted by WRITE operations.

A *merge policy* (also known as a *compaction* policy) determines how merges are done. The policy must efficiently trade off total write amplification for total read cost (which increases with the average SSTable count). This paper focuses on what we call *bounded depth* policies — those that guarantee a bounded number of disk accesses for each READ operation by ensuring that, at any given time, the SSTable count (the number of existing SSTables) never exceeds a given parameter k, typically 3–10. Maintaining bounded depth is important in applications that require low read latency, but bounded-depth policies are not yet well understood.

A recent theoretical work by Mathieu et al. [6] (including one of the current authors) formally defines a broad class of so-called *stack-based policies* (see Section IV for the definition). This class includes policies of many popular NoSQL systems, including Bigtable [2], HBase [7], [8], [9], Accumulo [10], [9], Cassandra [11], Hypertable [12], and AsterixDB [13]. In contrast, *leveled* policies (used by LevelDB and its spin-offs [14]) split SSTables by key-space to avoid monolithic merges, so they do not fit the stack-based model. Note that all current leveled implementations yield unbounded depth, hence they are not considered here.

Mathieu et al. also propose theoretical metrics for policy evaluation, and, as a proof of concept, propose new policies that, among stack-based policies, are optimal according to those metrics. Two such policies, MINLATENCY and BINOMIAL (defined in Section III) are bounded-depth policies designed to have minimum *worst-case write amplification* (subject to the depth constraint) among all stack-based policies. Mathieu et al. observe that, according to the theoretical model, on some inputs *existing policies are far from optimal*, so, on some common workloads, compared to existing policies, MINLATENCY and BINOMIAL can have lower write amplification.

Here we empirically compare MINLATENCY and BINOMIAL to representative bounded-depth policies from state-of-the-art NoSQL databases: a policy from AsterixDB [15], EXPLORING (the default policy for Apache HBase [16]), and the default policy from Bigtable (as described by Mathieu et al. [6], which includes authors from Google). Section III defines these policies. We implement the policies under con-

sideration on a common platform — Apache AsterixDB [13], [15], — and evaluate them on inputs from the Yahoo! Cloud Serving Benchmark (YCSB) [17], [18]. This is the first implementation and evaluation of the policies proposed by Mathieu et al. on a real NoSQL system. The empirical results validate the theoretical model. MINLATENCY and BINOMIAL achieve write amplification close to the theoretical minimum, thereby outperforming the other policies by orders of magnitude on some realistic workloads. (See Section V.)

Having a realistic theoretical model facilitates merge-policy design both via theoretical analysis (as for MINLATENCY and BINOMIAL), and because it enables rapid but faithful simulation of experiments. NoSQL systems are designed to run for months, incorporating hundreds of terabytes. Experiments can take weeks, even with appropriate adaptations. In contrast, the model allows some experiments to be faithfully simulated in minutes. (See Section VI.)

In summary, this work makes the following contributions:

1) The implementation of several existing and recently proposed merge policies on a common, open-source platform, specifically Apache AsterixDB.
2) An experimental evaluation confirming that the recently proposed policies can significantly outperform the state-of-the-art policies on some common workloads.
3) An empirical validation of a realistic cost model, which facilitates the design of merge policies via theoretical analysis and rapid simulation.

II. RELATED WORK

Historically, the main data structure used for on-disk key-value storage is the B^+-tree. Nonetheless, LSM architectures are becoming common in industrial settings. This is partly because they offer substantially better performance for write-heavy workloads [19]. Further, for many workloads, reads are highly cacheable, making the *effective* workload write-heavy. In these cases, LSM architectures substantially outperform B^+-trees.

In 2006 Google released Bigtable [20], [2], now the primary data store for many Google applications. Its default merge policy is a bounded-depth stack-based policy. We study it here. Spanner [21], Google's Bigtable replacement, likely uses a stack-based policy, though details are not public.

Apache HBase [16], [8], [7] was introduced around 2006, modeled on Bigtable, and used by Facebook 2010–2018. Its default merge policy is EXPLORING, the precursor of which was RATIOBASED, a variant of BIGTABLE. Both policies are configurable as bounded-depth policies. Here we report results only for EXPLORING, as it consistently outperformed RATIOBASED.

Apache Cassandra [22], [11] was released by Facebook in 2008. Its first main merge policy, SIZETIERED, is a stack-based policy that orders the SSTables by length, groups similar-length SSTables, and then merges a group that has sufficiently many SSTables. SIZETIERED is not *stable* — that is, it does not maintain the following property at all times: *the* WRITE *times of all items in any given SSTable precede those of all items in every newer SSTable.* With a stable policy, a READ can scan the recently created SSTables first, stopping with the first SSTable that contains the key. Unstable policies lack this advantage: a READ operation must check *every* SSTable. Apache Accumulo [10], which was created in 2008 by the NSA, uses a similar stack-based policy. We don't test these policies here, as our test platform supports only stable policies, and we believe they behave similarly to BIGTABLE or EXPLORING.

Previous to this work, our test platform — Apache AsterixDB — provided just one bounded-depth policy (CONSTANT), which suffered from high write amplification [23]. AsterixDB has removed support for CONSTANT, and, based on the preliminary results here, added support for BINOMIAL.

Leveled policies: LevelDB [14], [24] was released by Google in 2011. Its merge policy, unlike the policies mentioned above, does not fit the stack-based model. For our purposes, the policy can be viewed as a modified stack-based policy where each SSTable is split (by partitioning the key space into disjoint intervals) into multiple smaller SSTables that are collectively called a *level* (or *sorted run*). Each READ operation needs to check only one SSTable per level — the one whose key interval contains the given key. Using many smaller tables allows smaller, "rolling" merges, avoiding the occasional monolithic merges required by stack-based policies (but see [25, §3.2]).

In 2011, Apache Cassandra added support for a leveled policy adapted from LevelDB. (Cassandra also offers merge policies specifically designed for time-series workloads.) In 2012, Facebook released a LevelDB fork called RocksDB [26], [27]. RocksDB offers several policies: one modelled on LevelDB, UNIVERSALCOMPACTION (a stack-based policy similar to Cassandra's SIZETIERED), and hybrids.

None of the leveled policies are bounded-depth policies.

Other merge-policy models and optimizations: Independently of Mathieu et al. [6], Lim et al. [28] propose a similar theoretical model for write amplification and point out its utility for simulation. The model includes a statistical estimate of the effects of for UPDATEs and DELETEs. For leveled policies, Lim et al. use their model to propose tuning various policy parameters — such as the size of each level — to optimize performance. Dayan et al. [5], [29] propose further optimizations of SIZETIERED and leveled policies by tuning aspects such as the Bloom filters' false positive rate (vs. size) according to SSTable size, the per-level merge frequency, and the memory allocation between buffers and Bloom filters.

Multi-threaded merges (exploiting SSD parallelism) are studied in [30], [31], [27], [32]. Cache optimization in leveled merges is studied in [33]. Offloading merges to another server is studied in [34].

Some of the methods above optimize READ performance; those complement the optimization of write amplification considered here. None of the above works consider bounded-depth policies.

This paper focuses primarily on write amplification (and to some extent read amplification). Other aspects of LSM performance, such as space amplification, can also be affected by merge policies but are not discussed here. For a more detailed discussion of LSM architectures, including compaction policies, see [35].

III. Policies Studied in this Paper

Bigtable (Google): The default for the Bigtable platform is as follows [6]. *When the MemTable is flushed, if there are fewer than k SSTables, add a single new SSTable holding the MemTable contents. Otherwise, merge the MemTable with the i most recently created SSTables, where i is the minimum such that, afterwards, the length of each SSTable exceeds the sum of the lengths of all newer SSTables.*[1] Roughly speaking, this ensures that each SSTable is at most half the length of the next older SSTable. We denote this policy BIGTABLE.

Exploring (Apache HBase): EXPLORING is the default for HBase [16]. In addition to $\boldsymbol{k}$, it has configurable parameters $\boldsymbol{\lambda}$ (default 1.2), $\boldsymbol{C}$ (default 2), and $\boldsymbol{D}$ (default 10). When the MemTable (Memstore in HBase) is flushed, the policy orders the SSTables (HFiles in HBase) by time of creation, considers various contiguous subsequences of them, and merges one that is in some sense most cost-effective. Specifically: *Temporarily add the MemTable as its own (newest) SSTable, then consider every contiguous subsequence s such that*

1) *s has at least C and at most D SSTables, and*
2) *in s, the length of the largest SSTable is at most λ times the sum of the lengths of the other SSTables.*

In the case that there is at least one such subsequence s, merge either the longest (if there are at most k SSTables) or the one with minimum average SSTable length (otherwise). In the remaining case, and only if there are more than k SSTables, merge a contiguous subsequence of C SSTables having minimum total length.

Constant (AsterixDB before version 0.9.4): CONSTANT is as follows. *When the MemTable is flushed, if there are fewer than k SSTables, add a single new SSTable holding the MemTable contents. Otherwise, merge the MemTable and* all *k SSTables into one.*

Next are the (somewhat cryptic!) definitions of the MINLATENCY and BINOMIAL policies proposed by Mathieu et al. First, define a utility function B, as follows. Consider any binary search tree T with nodes $\{1, 2, \ldots, n\}$ in search-tree order (each node is larger than those in its left subtree, and smaller than those in its right subtree). Given a node t in T, define its *stack (merge) depth* to be the number of ancestors smaller (larger) than t. (Hence, the depth of t in T equals its stack depth plus its merge depth.)

Fix any two positive integers k and m, and let $n = \binom{m+k}{k} - 1$. Let $\tau^*(m, k)$ be the unique n-node binary search tree on nodes $\{1, 2, \ldots, n\}$ that has maximum stack depth $k - 1$ and maximum write depth $m - 1$. For $t \in \{1, 2, \ldots, n\}$, define $B(m, k, t)$ to be the stack depth of node t in T.

Compute the function $B(m, k, t)$ via the following recurrence. Define $B(m, k, 0)$ to be zero, and for $t > 0$ use

$$B(m,k,t) = \begin{cases} B(m-1,k,t) & \text{if } t < \binom{m+k-1}{k}, \\ 1 + B\Big(m, k-1, t - \binom{m+k-1}{k}\Big) & \text{if } t \geq \binom{m+k-1}{k}. \end{cases}$$

The policies are defined as follows.

MinLatency (Mathieu et al.): For each $t = 1, 2, \ldots,$ in response to the tth flush, the action of the policy is determined by t, as follows:

Let $m' = \min\{m : \binom{m+k}{m} > t\}$ and $i = B(m', k, t)$. Order the SSTables by time of creation, and merge the ith oldest SSTable with all newer SSTables and the flushed MemTable (leaving i SSTables).

Binomial (Mathieu et al.): For each $t = 1, 2, \ldots,$ in response to the tth flush, the action of the policy is determined by t, as follows:

Let $T_k(m) = \sum_{i=1}^{m} \binom{i+\min(i,k)-1}{i}$ and $m' = \min\{m : T_k(m) \geq t\}$.

Let $i = 1 + B(m', \min(m', k) - 1, t - T_k(m' - 1) - 1)$. Order the SSTables by time of creation, and merge the ith oldest SSTable with all newer SSTables and the flushed MemTable (leaving i SSTables).

As described in Section IV, these policies are designed carefully to have the minimum possible *worst-case write amplification* among all policies in the aforementioned class of stack-based policies.

BIGTABLE and (although it is not obvious from its specification) MINLATENCY are *lazy* — whenever the MemTable is flushed, if there are fewer than k SSTables, the policy leaves those SSTables unchanged, and creates a new SSTable holding just the flushed MemTable's contents. For this reason, these policies tend to keep the number of SSTables close to k. In contrast, for moderate-length runs (4^k or fewer flushes, as discussed later), EXPLORING and BINOMIAL often merge multiple SSTables even when fewer than k SSTables are already present, so may keep the average number of SSTables well below k, potentially allowing faster READ operations.

IV. Design of MinLatency and Binomial

This section reviews Mathieu et al's definition of the class of so-called *stack-based* merge policies, the *worst-case write amplification* metric, and how MINLATENCY and BINOMIAL are designed to minimize that metric among all policies in that class.

A. Bounded-depth stack-based merge policies

Informally, a *stack-based policy* must maintain a set of SSTables over time. The set is initially empty. At each time $t = 1, 2, \ldots, n$, the MemTable is flushed, having current length in bytes equal to a given integer $\ell_t \geq 0$. In response, the merge policy must choose some of its current SSTables, then replace those chosen SSTables by a single SSTable holding their contents and the MemTable contents. As a special case, the policy may create a new SSTable from the MemTable

[1] The implementation of this and other policies may temporarily create an SSTable holding the MemTable contents, and then merge that SSTable with the other SSTables.

contents alone. (The policy may replace additional sets of SSTables by their respective unions, but the policies studied here don't.)

Each newly created SSTable is written to disk, batch-writing a number of bytes equal to its length, which *by assumption* is the sum of the lengths of the SSTables it replaces, plus ℓ_t if the merge includes the flushed MemTable. (This ignores UPDATEs and DELETEs, but see the discussion below.)

A *bounded-depth* policy (in the context of a parameter k) must keep the SSTable count at k or below. Subject to that constraint, its goal is to minimize the *write amplification*, which is defined to be the total number of bytes written in creating SSTables, divided by $\sum_{t=1}^{n} \ell_t$, the sum of the lengths of the n MemTable flushes. (Write amplification is a standard measure in LSM systems [27], [28], [32], [25].)

For intuition, consider the example $k = 2$ and $\ell_t = 1$ uniformly for $t \in \{1, 2, \ldots, n\}$. The optimal write amplification is $\Theta(\sqrt{n})$.

Next is the precise formal definition, as illustrated in Figure 1(a):

Problem 1 (k-Stack-Based LSM Merge): A problem instance is an $\ell \in \mathbb{R}_+^n$. For each $t \in \{1, \ldots, n\}$, say *flush* t has *(flush) length* ℓ_t. A solution is a sequence $\sigma = \{\sigma_1, \ldots, \sigma_n\}$, called a *schedule*, where each σ_t is a partition of $\{1, 2, \ldots, t\}$ into at most k parts, each called an *SSTable*, such that σ_t is refined by[2] $\sigma_{t-1} \cup \{\{t\}\}$ (if $t \geq 2$). The *length* of any SSTable F is defined to be $\ell(F) = \sum_{t \in F} \ell_t$ — the sum of the lengths of the flushes that comprise F. The goal is to minimize σ's *write amplification*, defined as $W(\sigma) = \sum_{t=1}^{n} \delta(\sigma_t, \sigma_{t-1}) / \sum_{t=1}^{n} \ell_t$, where $\delta(\sigma_t, \sigma_{t-1}) = \sum_{F \in \sigma_t \setminus \sigma_{t-1}} \ell(F)$ is the sum of the lengths of the new SSTables created during the merge at time t.

Formally, a *(bounded-depth) stack-based merge policy* is a function P mapping each problem instance $\ell \in \mathbb{R}_+^n$ to a solution σ. In practice, the policy must be *online*, meaning that its choice of merge at time t depends only on the flush lengths $\ell_1, \ell_2, \ldots, \ell_t$ seen so far. Because future flush lengths are unknown, no online policy P can achieve minimum possible write amplification for *every* input ℓ. Among possible metrics for analyzing such a policy P, the focus here is on *worst-case* write amplification: the maximum, over all inputs $\ell \in \mathbb{R}_+^n$ of length n, of the write amplification that P yields on the input. Formally, this is the function $n \mapsto \max\{W(P(\ell)) : \ell \in \mathbb{R}_+^n\}$.

Updates and Deletes: The formal definitions above ignore the effects of key UPDATEs and DELETEs. While it would not be hard to extend the definition to model them, for designing policies that minimize worst-case write amplification, this is unnecessary: these operations only *decrease* the write amplification for a given input and schedule, so any online policy in the restricted model above can easily be modified to achieve the same worst-case write amplification, even in the presence of UPDATEs and DELETEs.

Additional terminology: Recall that a policy is *stable* if, for every input, it maintains the following invariant at all times among the current SSTables: *the* WRITE *times of all items in any given SSTable precede those of all items in every newer SSTable.* (Formally, every SSTable created is of the form $\{i, i+1, \ldots, j\}$ for some i, j.) As discussed previously, this can speed up READs. We note without proof that any unstable solution can be made stable while at most doubling the write amplification. Likewise, each uniform input has an optimal stable solution. All policies tested here are stable.

A policy is *eager* if, for every input ℓ, for every time t, the policy creates just one new SSTable (necessarily including the MemTable flushed at time t). Every input has an optimal eager solution, and all policies tested here except for EXPLORING are eager.

An online policy is *static* if each σ_t is determined *solely by k and t*. In a static policy, the merge at each time t is predetermined — for example, for $t = 1$ merge just the flushed MemTable, for $t = 2$ merge the MemTable with the top SSTable, and so on — independent of the flush lengths $\ell_1, \ell_2, \ldots$ The MINLATENCY and BINOMIAL policies are static. Static policies ignore the flush lengths, so it may seem counter-intuitive that static policies can achieve optimum worst-case write amplification.

B. MinLatency and Binomial

Among bounded-depth stack-based policies, MINLATENCY and BINOMIAL, by design, have the minimum possible worst-case write amplification. Their design is based on the following relationship between schedules and binary search trees.

Fix any k-Stack-based LSM Merge instance $\ell = (\ell_1, \ldots, \ell_n)$. Consider any eager, stable schedule σ for ℓ. (So σ creates just one new SSTable at each time t.) Define the (rooted) *merge forest* $\mathcal{F}$ for σ as follows: for $t = 1, 2, \ldots, n$, represent the new SSTable F_t that σ creates at time t by a new node t in $\mathcal{F}$, and, for each SSTable F_s (if any) that is merged in creating F_t, make node t the parent of the node s that represents F_s.

Next, create the *binary search tree* T for σ from $\mathcal{F}$ as follows. Order the roots of $\mathcal{F}$ in decreasing order (decreasing creation-time t). For each node in $\mathcal{F}$, order its children likewise. Then let $T = T(\sigma)$ be the standard left-child, right-sibling binary tree representation of $\mathcal{F}$. That is, T and $\mathcal{F}$ have the same vertex set $\{1, 2, \ldots, n\}$, and, for each node t in T, the left child of t in T is the first (oldest) child of t in $\mathcal{F}$ (if any), while the right child of t in T is the right (next oldest) sibling of t in $\mathcal{F}$ (if any; here we consider the roots to be siblings). It turns out that (because σ is stable) the nodes of T must be in search-tree order (each node is larger than those in its left subtree and smaller than those in its right subtree). Figures 1(a) and 1(c) give an example.

What about the depth constraint on σ, and its write amplification? Recall that the *stack (merge) depth* of a node t is the number of ancestors that are smaller (larger) than t. While the details are out of scope here, the following holds:

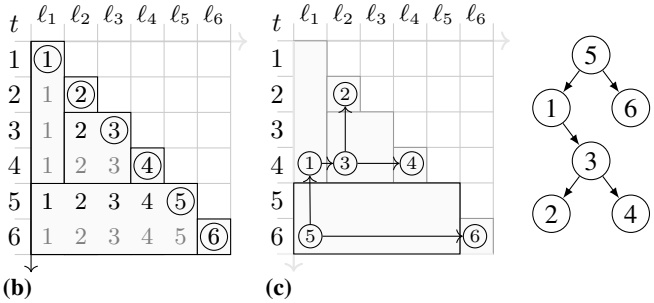
For any eager, stable schedule σ:

1) *σ obeys the depth constraint if and only if every node in $T(\sigma)$ has stack depth at most $k - 1$,*

[2]Each part in σ_t is the union of some parts in $\sigma_{t-1} \cup \{\{t\}\}$.

t	SSTables at time t	bytes written	SSTable count
1	$\sigma_1 = \{\mathbf{1}\}$	ℓ_1	1
2	$\sigma_2 = \{1\}, \{\mathbf{2}\}$	ℓ_2	2
3	$\sigma_3 = \{1\}, \{\mathbf{2}, \mathbf{3}\}$	$\ell_2 + \ell_3$	2
4	$\sigma_4 = \{1\}, \{2, 3\}, \{\mathbf{4}\}$	ℓ_4	3
5	$\sigma_5 = \{\mathbf{1}, \mathbf{2}, \mathbf{3}, \mathbf{4}, \mathbf{5}\}$	$\ell_1 + \ell_2 + \ell_3 + \ell_4 + \ell_5$	1
6	$\sigma_6 = \{1, 2, 3, 4, 5\}, \{\mathbf{6}\}$	ℓ_6	2

(a)



(b) **(c)**

Fig. 1: **(a)** An eager, stable schedule σ ($n = 6$, $k = 3$). **(b)** A graphical representation of σ. Each shaded rectangle is an SSTable (over time). Row t is the stack at time t. **(c)** The binary-search-tree representation of σ.

2) *the write amplification incurred by σ on ℓ equals* $\frac{\sum_{t=1}^{n}(\mathsf{mergedepth}(t,T(\sigma))+1)\ell_t}{\sum_{t=1}^{n}\ell_t} \leq 1 + \max_{t=1}^{n} \mathsf{mergedepth}(t, T(\sigma))$.

The mapping $\sigma \to T(\sigma)$ is invertible. Hence, *any binary search tree t with nodes $\{1, 2, \ldots, n\}$, maximum stack depth $k-1$, and maximum merge depth $m-1$ yields a bounded-depth schedule σ (such that $T(\sigma) = t$), having write amplification at most m on any input $\ell \in \mathbb{R}_+^n$.*

Rationale for MinLatency: MINLATENCY uses this observation to produce its schedule [6]. First consider the case that $n = \binom{m+k}{k} - 1$ for some integer m. Among the binary search trees on nodes $\{1, 2, \ldots, n\}$, there is a unique tree with maximum stack depth $k-1$ and maximum merge depth $m-1$. Let $\tau^*(m, k)$ denote this tree, and let $\sigma^*(m, k)$ denote the corresponding schedule.

MINLATENCY is designed to output $\sigma^*(m, k)$ for any input of length n. Since $\tau^*(m, k)$ has maximum merge depth $m-1$, as discussed above, $\sigma^*(m, k)$ has write amplification at most m, which by calculation is

$$(1 + O(1/k))\, k\, n^{1/k}/c_k, \tag{1}$$

where $c_k = (k+1)/(k!)^{1/k} \in [2, e]$. This bound extends to arbitrary n, so MINLATENCY's worst-case write amplification is at most (1).

This is optimal, in the following sense: for every $\epsilon > 0$ and large n, no stack-based policy achieves worst-case write-amplification less than $(1 - \epsilon)k\, n^{1/k}/c_k$. This is shown by using the bijection described above to bound the minimum possible write amplification for *uniform* inputs.

Binomial and the small-n and large-n regimes: As mentioned previously, because MINLATENCY and BIGTABLE are lazy, they produce schedules whose average SSTable count is close to k. When n is large, any policy with near-optimal write amplification must do this. Specifically, in what we call the *large-n regime* — after the number of flushes exceeds 4^k or so — any schedule with near-optimal write amplification (e.g., for uniform ℓ) *must* have average SSTable count near k. In this regime, BINOMIAL behaves similarly to MINLATENCY. Consequently, in this regime, BINOMIAL still has minimum worst-case write amplification.

However, in what we call the *small-n regime* — until the number of flushes n reaches 4^k — it is possible to achieve near-optimal write-amplification while keeping the average SSTable count somewhat smaller. BINOMIAL is designed to do this [6]. In the small-n regime, it produces the schedule σ for the tree $\tau^*(m, m)$, for which the maximum stack depth and maximum merge depth are both $m \approx \log_2(n)/2$, so BINOMIAL's average SSTable count and write amplification are about $\log_2(n)/2$, which is at most k (in this regime) and can be less. Consequently, in the small-n regime, BINOMIAL can opportunistically achieve average SSTable count well below k. In this way it compares well to EXPLORING, and it behaves well even with unbounded depth ($k = \infty$).

V. EXPERIMENTAL EVALUATION

A. Test Platform: AsterixDB

Apache AsterixDB [13], [15] is a full-function, open-source Big Data Management System (BDMS). It has a shared-nothing architecture, with each node in an AsterixDB cluster managing one or more storage and index partitions for its datasets based on LSM storage. Each node uses its memory for a mix of storing MemTables of active datasets, buffering of file pages as they are accessed, and other memory-intensive operations. AsterixDB represents each SSTable as a B^+-tree, where the number of keys at each internal node is roughly the configured page size divided by the key size. (Internal nodes store keys but not values.) Secondary indexing is also available using B^+-trees, R-trees, and/or inverted indexes [23]. As secondary indexing is out of the scope of this paper, our experiments involve only primary indexes.

AsterixDB provides *data feed*s for rapid ingestion of data [36]. A *feed adapter* handles establishing the connection with a data source, as well as receiving, parsing and translating data from the data source into ADM objects [13] to be stored in AsterixDB. Several built-in feed adapters available for retrieving data from network sockets, local file system, or from applications like Twitter and RSS.

B. Experimental Setup

The experiments were performed on a machine with an Intel i3–4330 CPU running CentOS 7 with 8 GB of RAM and two mirrored (RAID 1) 1 TB hard drives. AsterixDB was configured to use 1 node controller, so all records are stored on the same disk location. The relatively small RAM size of 8 GB limits caching, to better simulate large workloads. The MemTable capacity was configured at 4 MB. The small MemTable capacity increases the flush rate to better simulate longer runs.

The workload was generated using the Yahoo! Cloud Serving Benchmark (YCSB) [17], [18], with default parameters. The full workload consists of 80,000,000 WRITEs, each writing one record with a primary key of 5 to 23 bytes plus 10 attributes of 100 bytes each, giving 11 attributes of about 1 KB total length. Each primary key is a string with a 4-byte prefix and a long integer (as a string).

To achieve high ingestion rate, we implemented a YCSB database-interface layer using the AsterixDB "socket_adapter" data feed (which retrieves data from a network socket) with an upsert data model, so that records are written lazily (without a duplicate key check). Upsert in AsterixDB is the equivalent of standard insert in other NoSQL systems, where, if an inserted record conflicts in the primary key with an existing record, it overwrites it.

The MemTable flushes were triggered by AsterixDB when the MemTable was near capacity, so the input ℓ generated by the workload was nearly uniform, with each flush length ℓ_t about 4 MB. This represents about 3,300 records per flush, so the input size n — the total number of flushes in the run — was just over $24,000$.

For each of the five merge policies tested, and for each $k \in \{3, 4, 5, 6, 7, 8, 10\}$, we executed a single run testing that policy, configured with that depth (SSTable count) limit k. All other policy parameters were set to their default values (see Section III). Each of the 35 runs started from an empty instance, then inserted all records of the workload into the database, generating just over 24,000 flushes for the merge policy.

For $k = 3$, some policies had significantly large write amplification and so did not finish the run. BINOMIAL and MINLATENCY finished in about 16 hours, but BIGTABLE and EXPLORING ingested less than 40% of the records after two days, so were terminated early. Similarly, CONSTANT was terminated early in *all* of its runs.

As our focus is on write amplification, which is not affected by READs, the workload contains no READs (but see Section V-D).

The data for all 35 runs is tabulated in Appendix A.

C. Policy Comparison (Write Amplification)

At any given time t during a run, define the *write amplification* (so far) to be the total number of bytes written to create SSTables so far divided by the number of bytes flushed so far ($\sum_{s=1}^{t} \ell_s$). This section illustrates how write amplification grows over time during the runs for the various policies. We focus on the runs with $k \in \{5, 6, 7\}$, which are particularly informative. The runs for each k are shown in Figures 2a–2c, each showing how the write amplification grows over the course of all $n \approx 24,000$ flushes. Because workloads with at most a few thousand flushes are likely to be important in practice, Figures 2d–2f repeat the plots, zooming in to focus on just the first $2,000$ flushes ($n = 2,000$).

In interpreting the plots, note that the caption of each subfigure shows the threshold 4^k. The small-n regime lasts until the number of flushes passes this threshold, whence the large-n regime begins. Note that (depending on n and k), some runs lie entirely within the small-n regime ($n \leq 4^k$), some show the transition, and in the rest (with $n \gg 4^k$) the small-n regime is too small to be seen clearly. In all cases, the results depend on the regime as follows. During the small-n regime, MINLATENCY has smallest write amplification, with BINOMIAL, BIGTABLE, and then EXPLORING close behind. As the large-n regime begins, MINLATENCY and BINOMIAL become indistinguishable. Their write amplification at time t grows sub-linearly (proportionally to $t^{1/k}$), while those of BIGTABLE and EXPLORING grow linearly (proportionally to t). These results are consistent with the analytical predictions from the theoretical model [6],

D. Policy Comparison (SSTable count)

As noted previously, MINLATENCY and BIGTABLE, being lazy, tend to keep the SSTable count near its limit k. In the large-n regime, any policy that minimizes worst-case write amplification must do this. But, in the small-n regime, BINOMIAL opportunistically achieves smaller average SSTable count, as does EXPLORING to some extent.

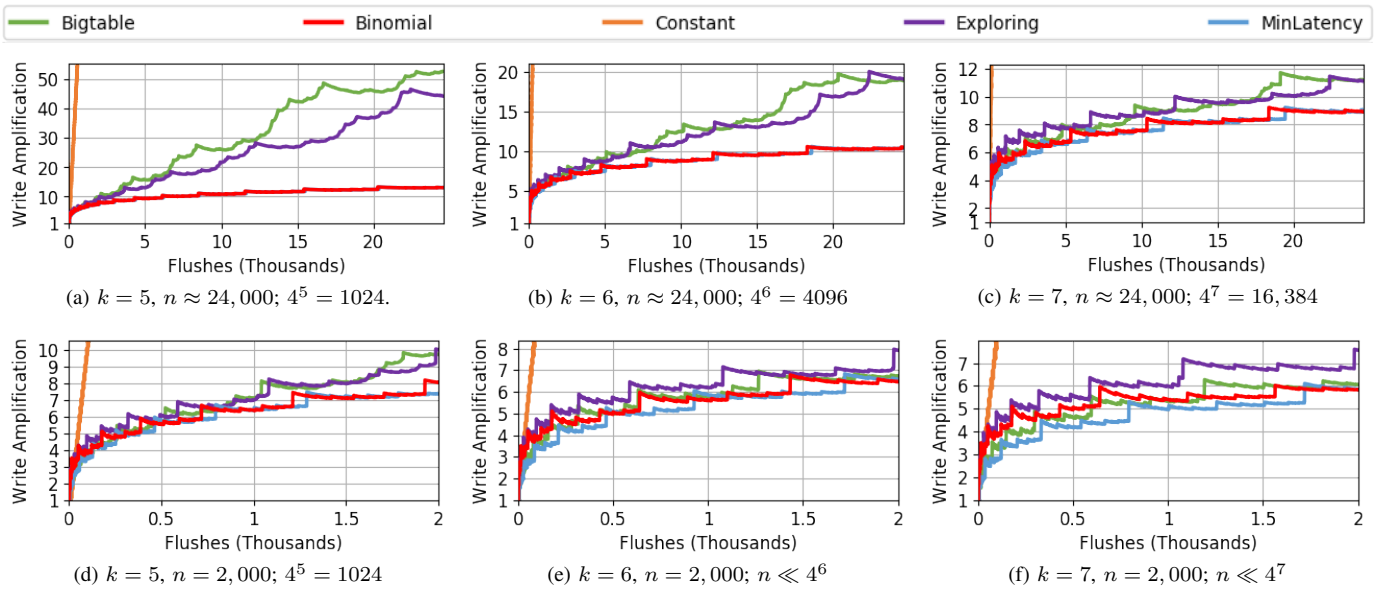
Figure 3 has a curve for each policy except CONSTANT. The curve shows the tradeoff between final write amplification and average SSTable count achieved by its policy: it has a point (x, y) for every run of the policy with $k \geq 4$ (and $n \approx 24,000$), where x is the final write amplification for the run and y is the average SSTable count over the course of the run. The x-axis is log-scaled.

First consider the runs with $k \in \{7, 6, 5, 4\}$. Within each curve, these correspond to the *four rightmost/lowest points* (with $k = 4$ being rightmost/lowest). These runs are dominated by the large-n regime, and each policies has average SSTable count (y coordinate) close to k. In this regime the BINOMIAL and MINLATENCY policies achieve the same (near-optimal) tradeoff, while the EXPLORING and BIGTABLE policies are far from the optimal frontier due to their larger write amplification.

Next consider the remaining runs, for $k \in \{10, 8\}$. On each curve, these are the two leftmost/highest points, with $k = 10$ leftmost. In the curve for EXPLORING, its two points are indistinguishable. These runs stay within the small-n regime. In this regime, BINOMIAL achieves a slightly better tradeoff than the other policies. MINLATENCY and BIGTABLE give comparable tradeoffs. For EXPLORING, its two runs lie close to the optimal frontier, but low: increasing the SSTable limit (k) from 8 to 10 makes little difference.

E. Updates and Deletions

UPDATE and DELETE operations insert records whose keys already occur in some SSTable. As a merge combines SSTables, if it finds multiple records with the same key, it can remove all but one. Hence, for workloads with UPDATE and DELETE operations, the write amplification can be reduced. But the experimental runs described above have no UPDATE or DELETE operations. As a step towards understanding their effects, we did additional runs with $k = 6$, with 70% of



(a) $k = 5$, $n \approx 24,000$; $4^5 = 1024$. (b) $k = 6$, $n \approx 24,000$; $4^6 = 4096$ (c) $k = 7$, $n \approx 24,000$; $4^7 = 16,384$

(d) $k = 5$, $n = 2,000$; $4^5 = 1024$ (e) $k = 6$, $n = 2,000$; $n \ll 4^6$ (f) $k = 7$, $n = 2,000$; $n \ll 4^7$

Fig. 2: Write amplification vs. number of flushes over time for runs with $k \in \{5, 6, 7\}$. The top row shows $n \approx 24,000$; the bottom shows $n = 2,000$. The transition from the small-n regime to the large-n regime (if present) occurs at 4^k flushes. Complete data is in Appendix A.

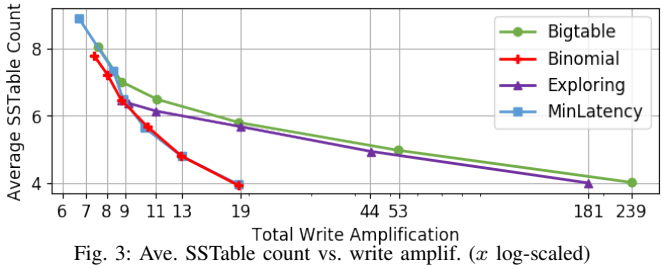


Fig. 3: Ave. SSTable count vs. write amplif. (x log-scaled)

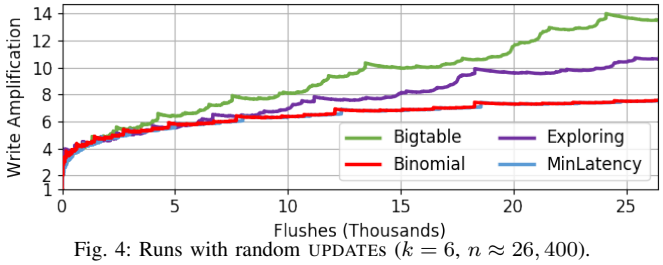


Fig. 4: Runs with random UPDATEs ($k = 6$, $n \approx 26,400$).

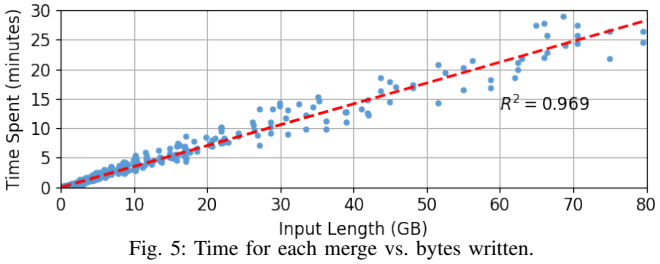


Fig. 5: Time for each merge vs. bytes written.

the WRITE operations replaced by UPDATEs, each to a key selected randomly from the existing keys according to a Zipf distribution with exponent $E = 0.99$, concentrated on the recently inserted items, similar to the "Latest" distribution in YCSB. The flush rate is reduced, as UPDATEs to keys currently in the MemTable are common but do not increase the number of records in the MemTable. To compensate, we increased the total number of operations by 50%, resulting in about $n \approx 26,400$ flushes.

Figure 4 plots the write amplification versus flushes for the four runs. The primary effect (not seen in the plots) is a reduction in the total number of flushes, but the write amplification (even as a function of the number of flushes) is also somewhat reduced, compared to the original runs (Figure 2b). The relative performance of the various policies is unchanged. Experiments with other key distributions (uniform over existing keys, or Zipf concentrated on the oldest) yielded similar results that we don't report here.

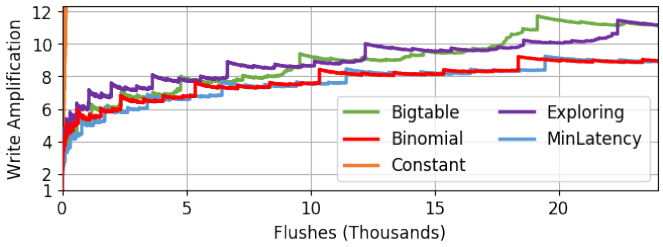
VI. Model Validation and Simulation

In each run, the *total time* spent in each merge operation is well-predicted by the bytes written. This is demonstrated by the plot in Figure 5, which has a point for *every individual merge* in every run, showing the elapsed time for that merge versus the number of bytes written by that merge.

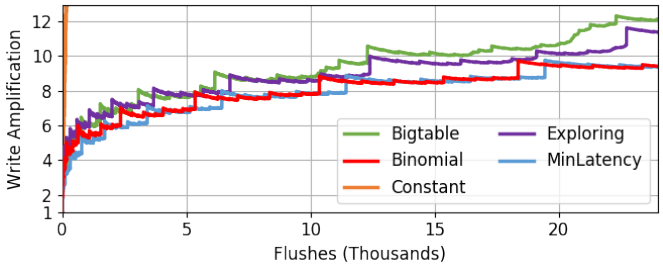
Also, observed write amplification is in turn well predicted by the theoretical model. More specifically, using the assumptions of the theoretical model, we implemented a simulator [37] that can simulate a given policy on a given input. For each of the 35 runs from the experiment, we simulated the run (for the given policy, k, and $n \approx 24,000$ uniform flushes).

Figure 6a illustrates the five runs with $k = 7$, over time, by showing the write amplifications over time as observed in the *actual* runs. Figure 6b shows the same for the *simulated* runs. For the static policies MINLATENCY and BINOMIAL, the observed write amplification tracks the predicted write amplification closely. For EXPLORING and BIGTABLE, the observed write amplification tracks the predicted write amplification, but not as closely. (For these policies, small perturbations in the flush lengths can affect total write amplification.)

Figure 7 shows that the simulated write amplification is a reasonable predictor of the write amplification observed



(a) Observed write amplification over time. ($k = 7, n = 24,000$)



(b) Simulated write amplification over time. ($k = 7, n = 24,000$)

Fig. 6: Observed and simulated write amplif. over time.

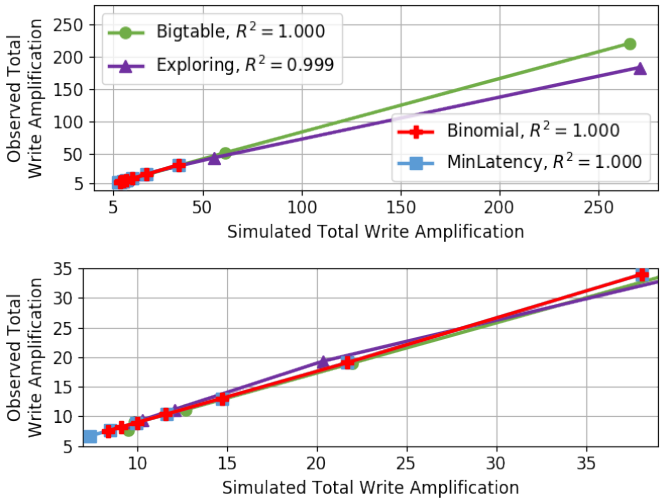


Fig. 7: Observed versus simulated write amplification. The bottom plot zooms in to the portion with $x \in [7, 39]$.

empirically. That figure has two plots. The first (top) plot in that figure has a curve for each policy (except CONSTANT), with a point for each of the six or seven runs that the policy completed, showing the observed final write amplification versus the simulated final write amplification. The two extreme points in the upper right are for BIGTABLE and EXPLORING with $k = 4$, with very high write amplification. To better show the remaining data, the second (bottom) plot expands the first, zooming in to the lower left corner (the region with $x \in [7, 39]$). For each curve, the R^2 value of the best-fit linear trendline is shown in the upper left of the first plot. (The trendlines are not shown.) The R^2 values are very close to 1, demonstrating that the simulated write amplification is a good predictor of the experimentally observed write amplification.

Policy design via analysis and simulation: A realistic theoretical model facilitates design in at least two ways. As described earlier, the model allows a precise theoretical analysis of the underlying combinatorics, as illustrated by the design of MINLATENCY and BINOMIAL. It also allows accurate simulation. As noted in the introduction, LSM systems are designed to run for months, incorporating terrabytes of data. Even with appropriate adaptations, real-world experiments can take days or weeks. Replacing experiments by (much faster) simulations can moderate this bottleneck. As a proof of concept, Figure 8 shows *simulated* write amplification over time for BIGTABLE, BINOMIAL, CONSTANT, EXPLORING, and MINLATENCY for $k \in \{6, 7, 10\}$. The first two plots show $k \in \{6, 7\}$ with $n = 100,000$ flushes. The third plot, with $k = 10$, shows one million flushes. These simulations took only minutes to complete.

VII. DISCUSSION

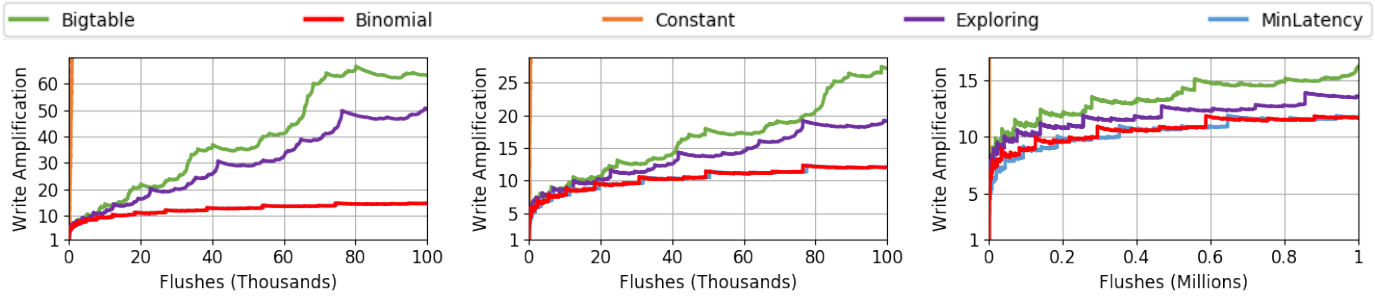
As predicted by the theoretical model, policy behavior fell into two regimes: the *small-n* regime (until the number of flushes reached about 4^k) and the *large-n* regime (after that). MINLATENCY achieved the lowest write amplification, with BINOMIAL a close second to it. BIGTABLE and EXPLORING were not far behind in the small-n regime, but in the large-n regime their write amplification was an order of magnitude higher. In short, the two newly proposed policies achieve near-optimal worst-case write amplification among all stack-based policies, outperforming policies in use in industrial systems, especially for runs with many flushes.

The tradeoffs between write amplification and average SSTable count were also studied. In the large-n regime, all policies except (sometimes) EXPLORING had average SSTable count near k. MINLATENCY and BINOMIAL, but not EXPLORING or BIGTABLE, were near the optimal frontier. In the small-n regime, all policies were close to the optimal frontier, with BINOMIAL and EXPLORING having average SSTable count below k.

Limitations and Future Work

Non-uniform flush lengths, UPDATE*s, dynamic policies.:* The experiments here are limited to near-uniform inputs, where most flush lengths are about the same. Workloads with near-uniform flush lengths are common, as most LSM systems routinely flush the MemTable when it reaches a configured capacity, but many systems trigger flushes for other reasons, so workloads with variable flush lengths are of interest. For moderately variable flush lengths, we expect the write amplification of most policies (and certainly of MINLATENCY and BINOMIAL) to be similar to that for uniform inputs. Regardless of variation, the write amplification incurred by MINLATENCY and BINOMIAL is guaranteed to be no worse than it is for uniform inputs.

Most of the experiments here are limited to APPEND-only workloads. A few preliminary results here suggest that a moderate rate of UPDATEs and DELETEs mainly reduce flush rate, and slightly reduce write amplification. At a minimum, UPDATEs and DELETEs are guaranteed not to increase the write amplification incurred by MINLATENCY and BINOMIAL. But inputs with UPDATEs, DELETEs, and non-uniform flush lengths



(a) $k = 6$, $n = 100,000$; $4^k = 4,096$ (b) $k = 7$, $n = 100,000$; $4^k = 16,384$ (c) $k = 10$, $n = 1,000,000$; $4^k \approx 10^6$

Fig. 8: Simulated total write amplification for 100,000 flushes (and 1,000,000 for $k = 10$).

can have optimal write amplification substantially below the worst case of $\Theta(n^{1/k})$. In this case, dynamic policies such as BIGTABLE, EXPLORING, and new policies designed using the theoretical framework of *competitive-analysis* (as in [6]), may, in principle, outperform static policies such as BINOMIAL and MINLATENCY. Future work will explore how significant this effect may be in practice.

Read costs: The experimental design here focuses on minimizing write amplification, relying on the bounded-depth constraint to control read costs such as *read amplification* — the average number of disk accesses required per read. Most LSM systems (other than Bigtable) offer merge policies that are not depth-bounded, instead allowing the SSTable count to grow, say, logarithmically with the number of flushes. A natural objective would be to minimize a linear combination of the read and write amplification — this could control the stack depth without manual configuration. (Similarly to BINOMIAL's behaviour in the small-n regime, where it minimizes the worst-case maximum of the read and write amplification, achieving a reasonable balance.) For read costs, a more nuanced accounting is desirable: It would be useful to take into account the effects of Bloom filters, and dynamic policies that respond to varying *rates* of READs are also of interest.

Leveled policies: The theoretical model that underlies MINLATENCY and BINOMIAL assumes that SSTable creation (while costly) is instantaneous, whereas, in practice, creating a large SSTable takes time, during which READs and WRITEs may occur.[3] Leveled policies (discussed previously) have the advantage of avoiding large SSTables and monolithic merges. It is of interest to extend the theoretical model to include leveled policies.

Secondary indexes: The existence of secondary indexes impacts merging. For example, AsterixDB (with default settings) maintains a one-to-one correspondence between the SSTables for the primary indexes and the SSTables for the secondary indexes. Ideally, merge policies should take secondary indexes into account.

VIII. CONCLUSIONS

This work compares several bounded-depth LSM merge policies, including representative policies from industrial NoSQL databases and two new ones based on recent theoretical modeling, on a common platform (AsterixDB) using Yahoo! cloud serving benchmark. The results validate the proposed theoretical model and show that, compared to existing policies, the newly proposed policies can have substantially lower write amplification. The theoretical model is realistic, and can be used, via both analysis and simulation, for the effective design and analysis of merge policies. For example, we shared our experimental findings with the developers of Apache AsterixDB [15], and BINOMIAL, designed via the theoretical model, has now been added as an LSM merging policy option to AsterixDB.

[3] As shown in the data table in Appendix A, for a few runs with $k \leq 4$, the average SSTable count exceeded k. This can happen when a flush occurs during the creation of a large SSTable.

ACKNOWLEDGMENT

We would like to thank the AsterixDB team for their help with the AsterixDB internals, and C. Staelin and A. Yousefi from Google for discussions about Bigtable. This work was supported by NSF grants IIS-1619463, IIS-1838222, IIS-1447826 and Google research award *A Study of Online Bigtable-Compaction Algorithms*.

A. APPENDIX (DATA)

For each of the 35 runs, Table I shows the total write amplification and average SSTable count at five points during the run: after $1,000$, $3,000$, $5,000$, $10,000$, and $20,000$ flushes. If it happens that the MemTable is flushed while a merge is ongoing, the SSTable count may briefly exceed k. For this reason, the average SSTable count slightly exceeded k in a few runs (with $k \in \{3, 4, 5\}$ — see the highlighted cells).

REFERENCES

[1] R. Cattell, "Scalable SQL and NoSQL data stores," *SIGMOD Rec.*, vol. 39, no. 4, pp. 12–27, May 2011.
[2] F. Chang *et al.*, "Bigtable: A distributed storage system for structured data," *ACM Trans. Comput. Syst.*, vol. 26, no. 2, pp. 4:1–4:26, Jun. 2008.
[3] P. O'Neil *et al.*, "The log-structured merge-tree (LSM-tree)," *Acta Inf.*, vol. 33, no. 4, pp. 351–385, Jun. 1996.
[4] G. Graefe *et al.*, "Modern B-tree techniques," *Foundations and Trends® in Databases*, vol. 3, no. 4, pp. 203–402, 2011.
[5] N. Dayan *et al.*, "Monkey: Optimal navigable key-value store," in *Proceedings of the 2017 ACM International Conference on Management of Data*, ser. SIGMOD, 2017, pp. 79–94.
[6] C. Mathieu *et al.*, "Bigtable merge compaction," *arXiv preprint arXiv:1407.3008*, vol. abs/1407.3008, 2014.
[7] L. George, *HBase: the definitive guide: random access to your planet-size data*. O'Reilly Media, Inc., 2011.

Policy	k	$n = 1,000$		$3,000$		$5,000$		$10,000$		$20,000$	
		Write Amplif.	SSTable Count	Write Amplif.	SSTable Count	Write Amplif.	SSTable Count	Write Amplif.	SSTable Count	Write Amplif.	SSTable Count
BIGTABLE	3	37.57	3.48	87.37	3.71	165.89	3.79	N/A			
	4	11.64	3.97	31.89	4.35	46.79	4.46	86.33	4.61	175.06	4.74
	5	7.13	4.53	11.19	4.79	15.80	4.96	26.16	5.17	46.17	5.36
	6	5.78	5.03	7.78	5.36	9.08	5.55	12.52	5.76	18.71	5.96
	7	5.34	5.60	6.69	5.92	7.85	6.10	9.22	6.31	11.46	6.52
	8	5.05	6.00	6.52	6.34	6.52	6.57	7.32	6.80	8.31	7.04
	10	5.31	6.97	5.79	7.39	6.63	7.51	7.26	7.73	7.87	7.98
BINOMIAL	3	12.05	2.95	17.26	3.01	20.61	3.03	25.72	3.08	32.07	3.13
	4	8.61	3.71	10.67	3.82	12.72	3.85	14.76	3.91	17.56	3.95
	5	6.38	4.49	8.84	4.65	9.34	4.70	10.86	4.77	12.49	4.83
	6	5.61	5.21	7.33	5.48	8.16	5.57	8.84	5.64	10.34	5.69
	7	5.40	5.39	6.44	6.05	6.77	6.24	7.57	6.40	8.96	6.49
	8	5.40	5.41	6.30	6.21	6.44	6.64	7.37	7.00	7.64	7.23
	10	5.40	5.38	6.30	6.19	6.44	6.62	7.30	7.08	7.19	7.68
EXPLORING	3	30.67	3.31	94.57	3.63	164.31	3.76	N/A			
	4	12.52	3.80	22.88	4.07	34.83	4.23	68.71	4.43	153.02	4.61
	5	7.00	4.31	10.55	4.66	13.08	4.79	21.72	5.06	37.00	5.26
	6	6.20	4.51	7.68	5.06	8.75	5.23	11.22	5.50	16.93	5.77
	7	5.99	4.58	7.41	5.19	7.73	5.42	8.66	5.81	10.07	6.12
	8	5.97	4.56	7.12	5.20	7.45	5.47	8.19	5.91	8.71	6.33
	10	5.90	4.55	6.99	5.19	7.33	5.48	7.98	5.97	8.37	6.40
MINLATENCY	3	12.10	3.00	17.26	3.03	20.62	3.04	25.72	3.08	32.07	3.12
	4	7.89	3.73	10.68	3.84	12.74	3.88	14.79	3.94	17.58	3.98
	5	6.38	4.51	8.11	4.66	9.37	4.70	10.90	4.76	12.48	4.82
	6	5.86	5.24	6.75	5.46	7.52	5.52	8.78	5.58	10.41	5.64
	7	4.97	6.09	5.96	6.30	6.59	6.37	7.55	6.44	9.13	6.51
	8	4.34	6.77	5.30	7.10	6.03	7.13	6.89	7.24	7.64	7.33
	10	3.69	8.24	4.52	8.56	5.03	8.63	5.87	8.78	6.93	8.91

TABLE I: Total observed write amplification and average SSTable count for all runs, for various n.

[8] A. Khetrapal *et al.*, "HBase and Hypertable for large scale distributed storage systems," *Dept. of Computer Science, Purdue University*, vol. 10, no. 1376616.1376726, 2006.

[9] S. Patil *et al.*, "YCSB++: Benchmarking and performance debugging advanced features in scalable table stores," in *Proceedings of the 2Nd ACM Symposium on Cloud Computing*, ser. SOCC '11. ACM, 2011, pp. 9:1–9:14.

[10] J. Kepner *et al.*, "Achieving 100,000,000 database inserts per second using Accumulo and D4M," in *2014 IEEE High Performance Extreme Computing Conference (HPEC)*, IEEE. IEEE, 2014, pp. 1–6.

[11] A. Lakshman *et al.*, "Cassandra: A decentralized structured storage system," *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, pp. 35–40, Apr. 2010.

[12] D. Judd, "Scale out with HyperTable," *Linux magazine, August 7th*, vol. 1, 2008. [Online]. Available: http://www.linux-mag.com/id/6645

[13] S. Alsubaiee *et al.*, "AsterixDB: A scalable, open source BDMS," *Proc. VLDB Endow.*, vol. 7, no. 14, pp. 1905–1916, Oct. 2014.

[14] "Google LevelDB." [Online]. Available: https://github.com/google/leveldb

[15] "Apache AsterixDB." [Online]. Available: https://asterixdb.apache.org

[16] "Apache HBase." [Online]. Available: https://hbase.apache.org

[17] "Yahoo! cloud serving benchmark." [Online]. Available: https://github.com/brianfrankcooper/YCSB

[18] B. F. Cooper *et al.*, "Benchmarking cloud serving systems with YCSB," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, ser. SoCC '10. ACM, 2010, pp. 143–154.

[19] C. Jermaine *et al.*, "The partitioned exponential file for database storage management," *The VLDB Journal*, vol. 16, no. 4, pp. 417–437, Oct. 2007.

[20] "Google Bigtable." [Online]. Available: https://cloud.google.com/bigtable

[21] J. C. Corbett *et al.*, "Spanner: Google's globally distributed database," *ACM Trans. Comput. Syst.*, vol. 31, no. 3, pp. 8:1–8:22, Aug. 2013.

[22] "Apache Cassandra." [Online]. Available: http://cassandra.apache.org

[23] S. Alsubaiee *et al.*, "Storage management in AsterixDB," *Proc. VLDB Endow.*, vol. 7, no. 10, pp. 841–852, Jun. 2014.

[24] A. Dent, *Getting started with LevelDB*. Packt Publishing Ltd, 2013.

[25] R. Sears *et al.*, "bLSM: A general purpose log structured merge tree," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '12. ACM, 2012, pp. 217–228.

[26] "Facebook RocksDB." [Online]. Available: https://rocksdb.org

[27] S. Dong *et al.*, "Optimizing space amplification in RocksDB." in *CIDR*, vol. 3. CIDR, 2017, p. 3.

[28] H. Lim *et al.*, "Towards accurate and fast evaluation of multi-stage log-structured designs," in *14th USENIX Conference on File and Storage Technologies (FAST 16)*. USENIX Association, 2016, pp. 149–166.

[29] N. Dayan *et al.*, "Dostoevsky: Better space-time trade-offs for LSM-Tree based key-value stores via adaptive removal of superfluous merging," in *Proceedings of the 2018 International Conference on Management of Data*, ser. SIGMOD '18. ACM, 2018, pp. 505–520.

[30] F. Chen *et al.*, "Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing," in *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, IEEE. IEEE, 2011, pp. 266–277.

[31] P. Wang *et al.*, "An efficient design and implementation of LSM-tree based key-value store on open-channel SSD," in *Proceedings of the Ninth European Conference on Computer Systems*, ser. EuroSys '14. ACM, 2014, pp. 16:1–16:14.

[32] L. Lu *et al.*, "WiscKey: Separating keys from values in SSD-conscious storage," *ACM Trans. Storage*, vol. 13, no. 1, pp. 5:1–5:28, Mar. 2017.

[33] D. Teng *et al.*, "LSbM-tree: Re-enabling buffer caching in data management for mixed reads and writes," in *Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on*, IEEE. IEEE, 2017, pp. 68–79.

[34] M. Y. Ahmad *et al.*, "Compaction management in distributed key-value datastores," *Proc. VLDB Endow.*, vol. 8, no. 8, pp. 850–861, Apr. 2015.

[35] C. Luo *et al.*, "LSM-based storage techniques: A survey," *CoRR*, vol. abs/1812.07527, 2018.

[36] R. Grover *et al.*, "Data ingestion in AsterixDB." in *EDBT*. OpenProceedings.org, 2015, pp. 605–616.

[37] "UCR Merge Simulator." [Online]. Available: https://github.com/UC-Riverside-DatabaseLab/MergeSimulator